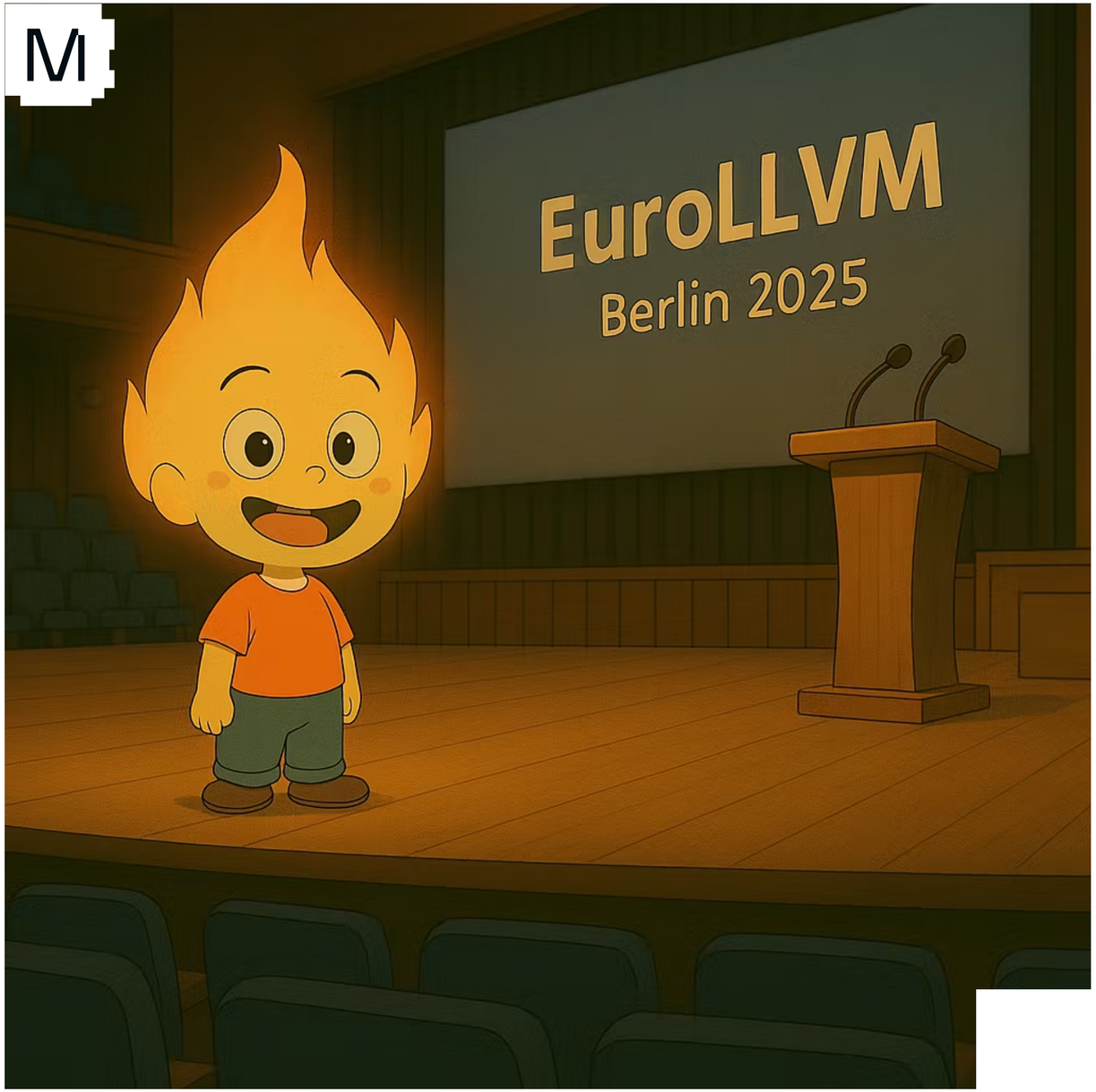
Modular

Parallelizing the LLVM Pipeline with MCLink 

Weiwei Chen
weiwei.chen@modular.com

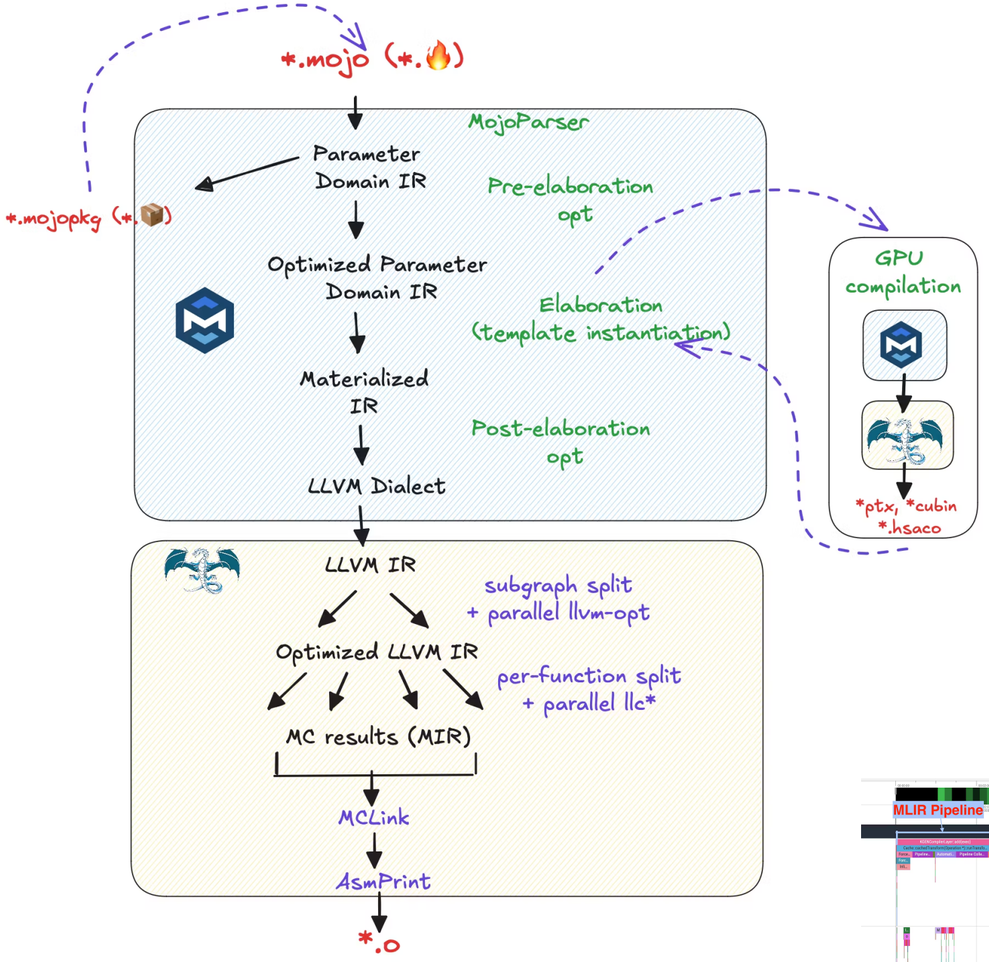
EuroLLVM 2025



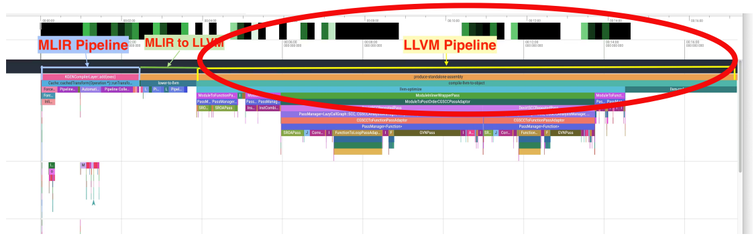
Agenda

01	Mojo Compilation Pipeline
02	Parallelize LLVM Compilation
03	MCLink
04	Mojo Compilation with Parallel LLVM
05	Conclusions

Mojo Compilation Pipeline



- Mojo compilation leverages the best of **MLIR** and **LLVM**
- Using LLVM unconventionally
  - Parallelizing LLVM pipeline.
  - Fast compilation time with performance generated code.
  - Significantly cut down LLVM pipeline time for overall mojo compilation.

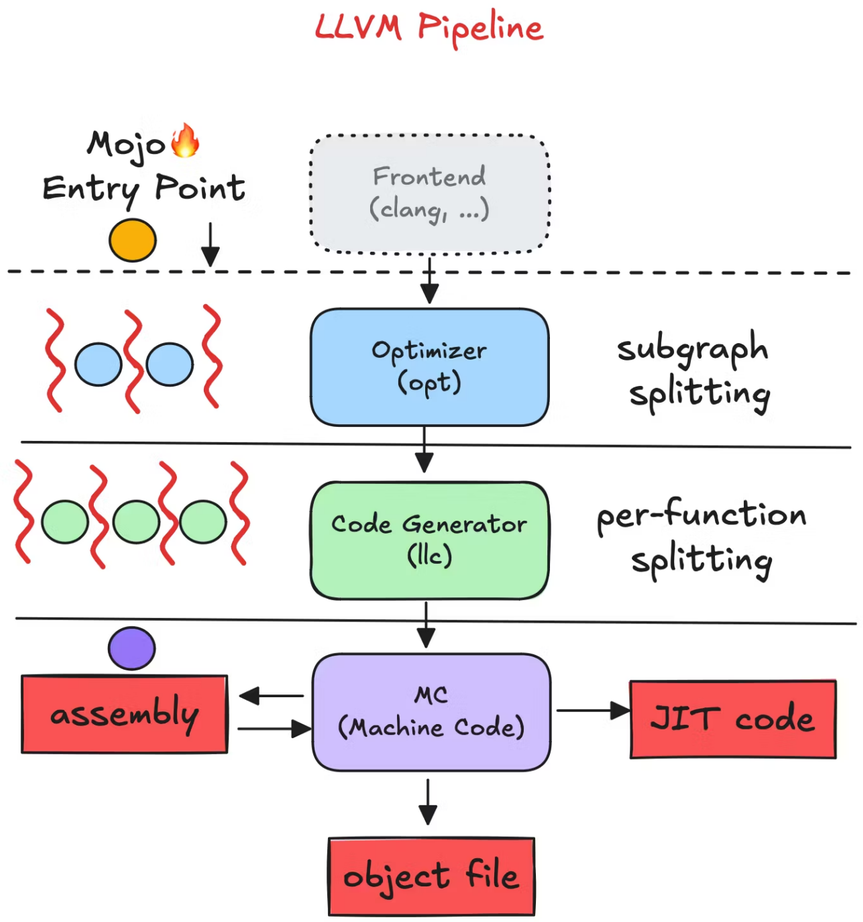


LLVM Pipeline

- LLVM is 🙂:
  - Target-specific code generation.
  - GVN, Load/Store Optimization, LSR, scalar optimization, etc.
- LLVM is 😞:
  - No framework-level parallelization support to leverage multi-core machines.
    - Data structures for IR are often mutable and not thread-safe.
    - No pass manager support for running function-level pass(es) in parallel.
    - Module–level passes don't have intra-pass parallelization.
  - Weak and unpredictable loop optimizations.
  - Often bottleneck for compilation time.

Parallelize LLVM Pipeline

- Split llvm::Module into multiple submodules.
  - Run each module split with separate LLVM pipeline in parallel.
- Two-level splitting:
  - Subgraphs for each anchor function (externally visible) with all function on its call-graph to run LLVM optimization pipeline including IPO passes (inliner).
  - Per-function split to run codegen pipeline in parallel.
- LLVM Pipeline:
  - Optimizer pipeline: llvm/lib/transforms
  - Code Generation pipeline: ISel, SelectDAG, RegAlloc, Machine Code opt, etc.
  - Machine Code layer: code emission

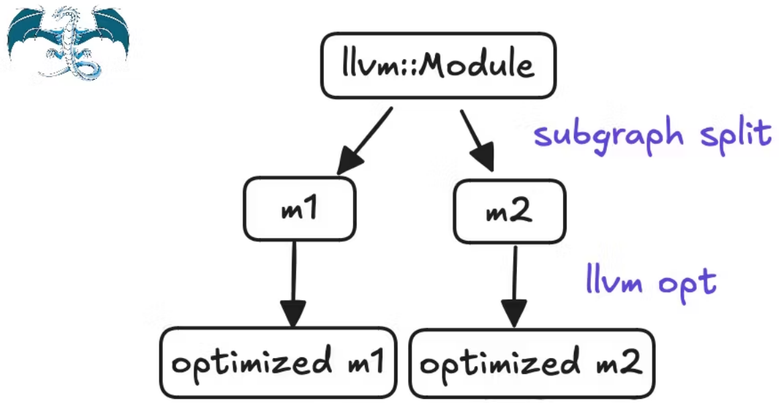


Parallelize LLVM Pipeline

llvm::Module

```
; llvm::Module
define dso_local void @foo() #0 {
  call void @g()
  call void @h()
  ret void
}
define dso_local void @bar() #0 {
  call void @g()
  call void @h()
  ret void
}
define internal void @g() #1 {
  ret void
}
define internal void @h() #1 {
  ret void
}
```

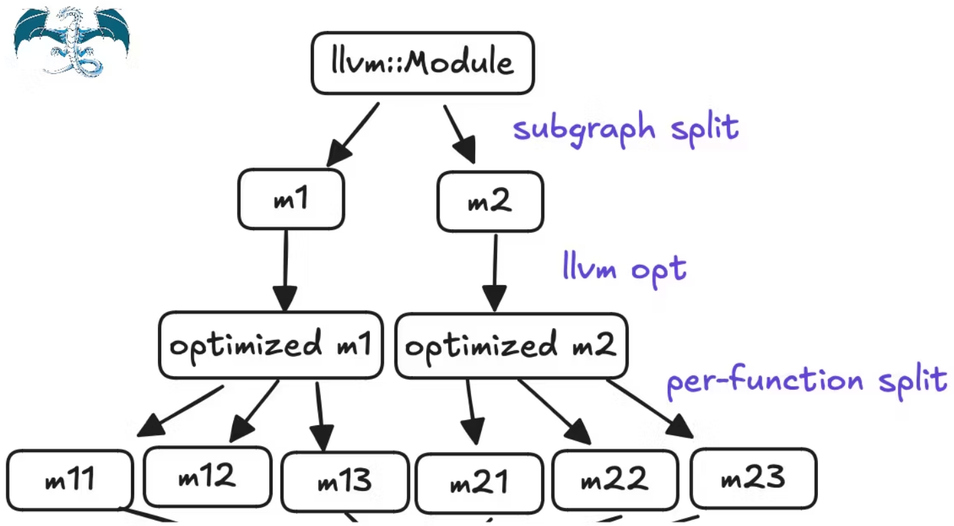
Parallelize LLVM Pipeline




```
; llvm::Module - m1
define dso_local void @foo() #0 {
  call void @g()
  call void @h()
  ret void
}
define internal void @g() #1 {
  ret void
}
define internal void @h() #1 {
  ret void
}
```

```
; llvm::Module - m2
define dso_local void @bar() #0 {
  call void @g()
  call void @h()
  ret void
}
define internal void @g() #1 {
  ret void
}
define internal void @h() #1 {
  ret void
}
```

Parallelize LLVM Pipeline




```
; llvm::Module - m11
define dso_local void @foo() #0 {
  call void @g()
  call void @h()
  ret void
}

declare void @g() #1
declare void @h() #1
```

```
; llvm::Module - m21
define dso_local void @bar() #0 {
  call void @g()
  call void @h()
  ret void
}

declare void @g() #1
declare void @h() #1
```

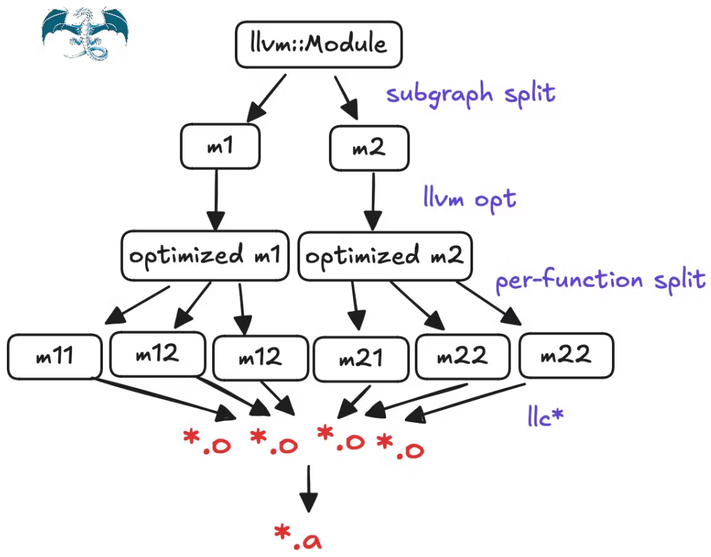
```
; llvm::Module - m12
define weak void @g() #1 {
  ret void
}
```

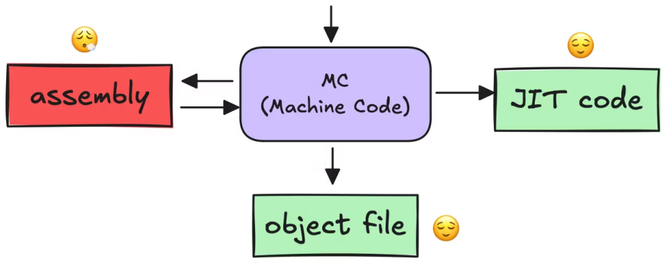
```
; llvm::Module - m22
define weak void @g() #1 {
  ret void
}
```

```
; llvm::Module - m13
define weak void @h() #1 {
  ret void
}
```

```
; llvm::Module - m23
define weak void @h() #1 {
  ret void
}
```

Parallelize LLVM Pipeline Code Emission

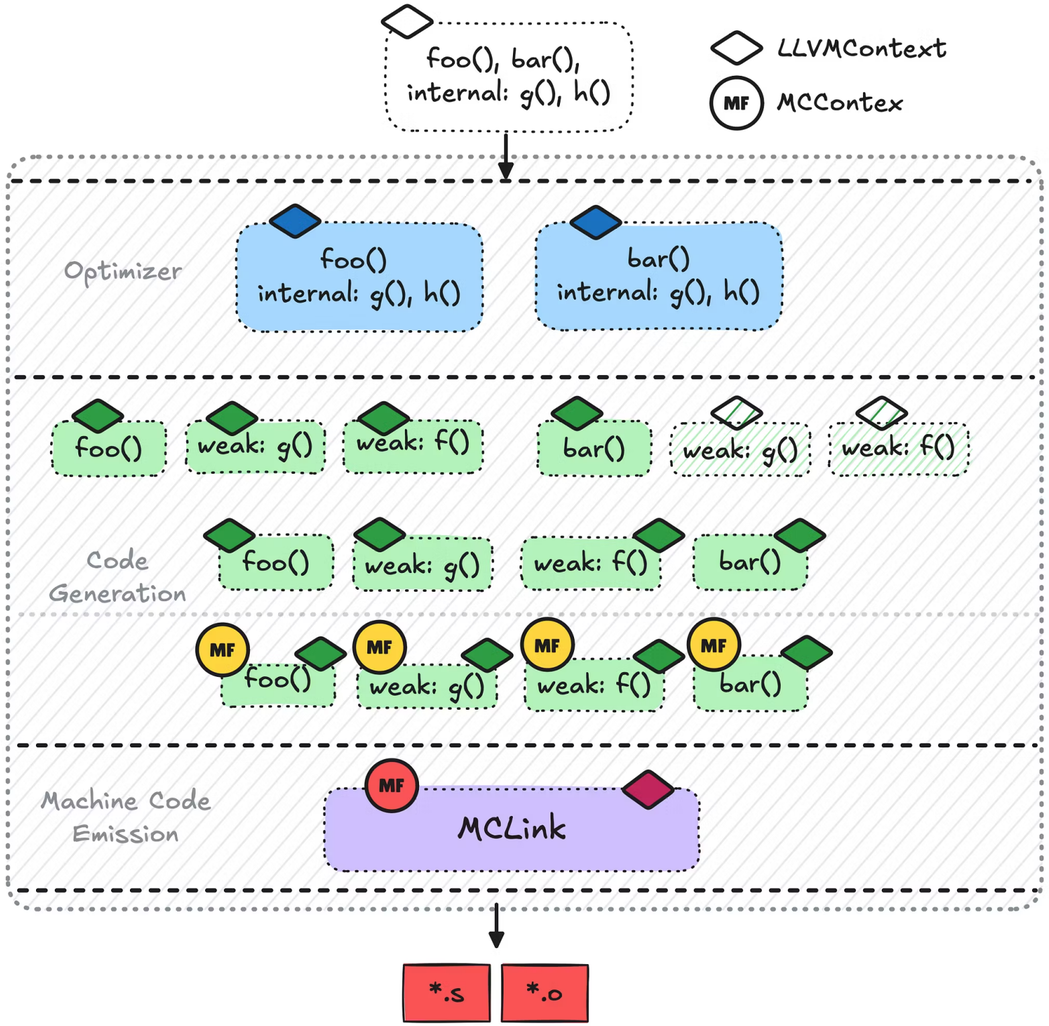




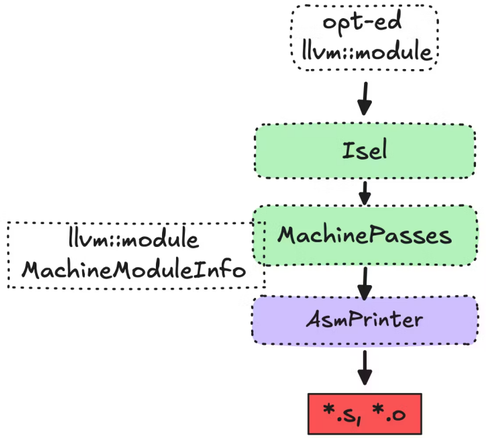
- Combine each split's codegen + code emission output
- .a archive file for binary object output (but bigger) ✅
- .a archive in buffer for JIT ✅
- Assembly output (debugging, PTX) ❌
- Can't fix symbol linkage type changes due to splitting (expose internal data structure and/or functions) ❌
- Duplicated functions 😞

Parallelize LLVM Pipeline Code Emission

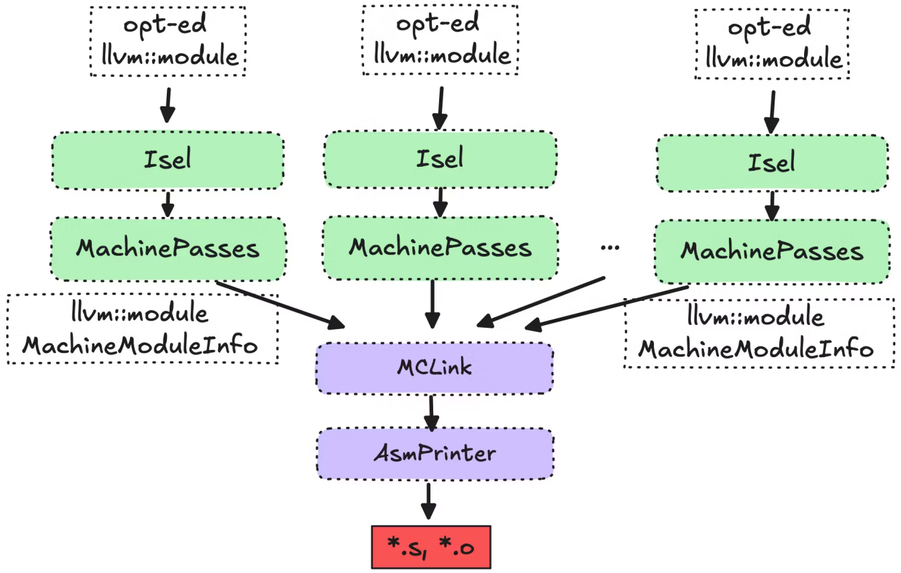
- 1 llvm::Module => 1 .o (.s)
  - Parallelization is implementation detail
  - No side-effect (symbol linkage change)
- Parallelization **maps** to multiple contexts
- MCLink **reduces** to one output



Linking at the MC level



LLVM CodeGen Pipeline

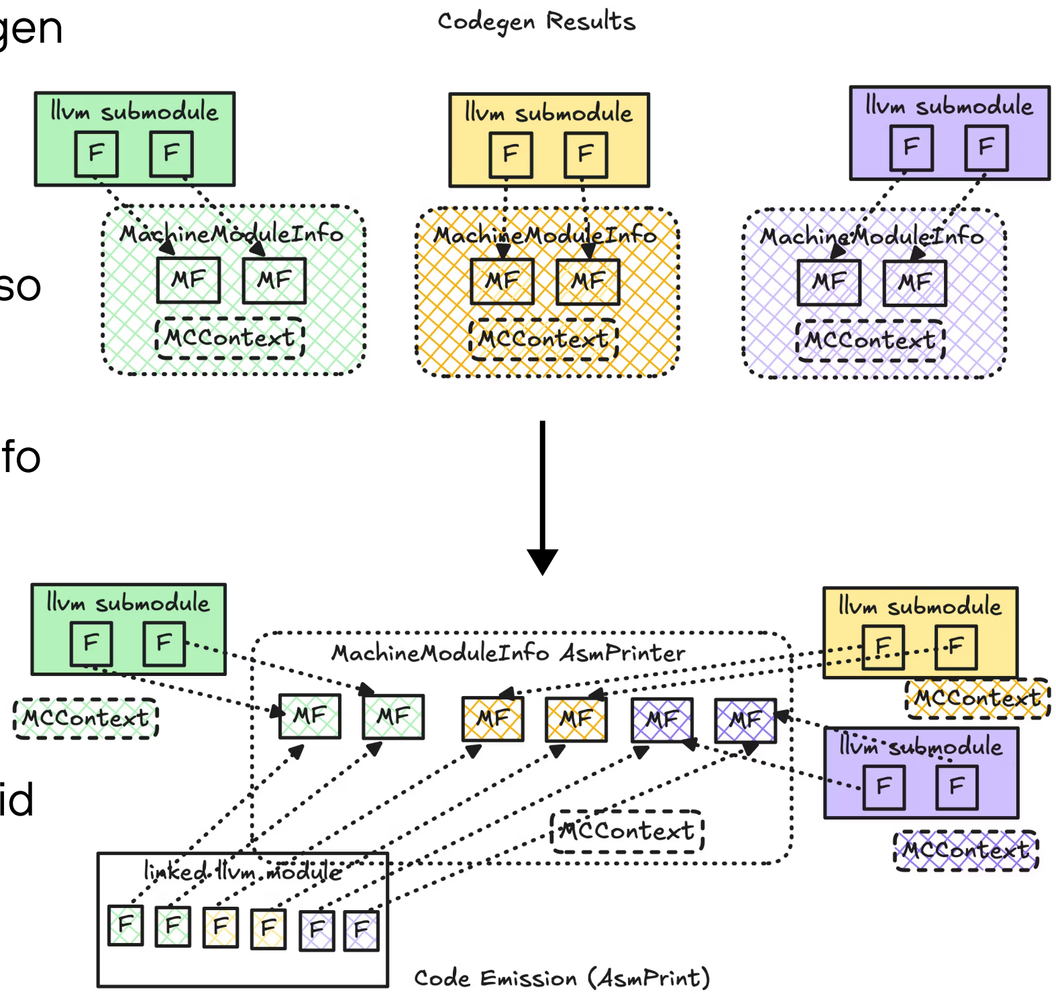


Parallel LLVM CodeGen Pipeline with MCLink

- Generate and Link MIR using llvm::Linker 😑
  - not quite stable?
  - YAML-based
- **MCLink** - Linker at the MC level 😌
  - Manage MC data structure for reduction
    - ConstantPool ID uniquing
    - X86:
      - PICBaseSymbol uniquing
      - MO_ExternalSymbol uniquing
  - Link llvm::Module from each split using llvm::Linker
    - Fix symbol linkage type
    - Reduces both LLVMContext and MCContext
    - Guide AsmPrinter

MCLink Implementation Details

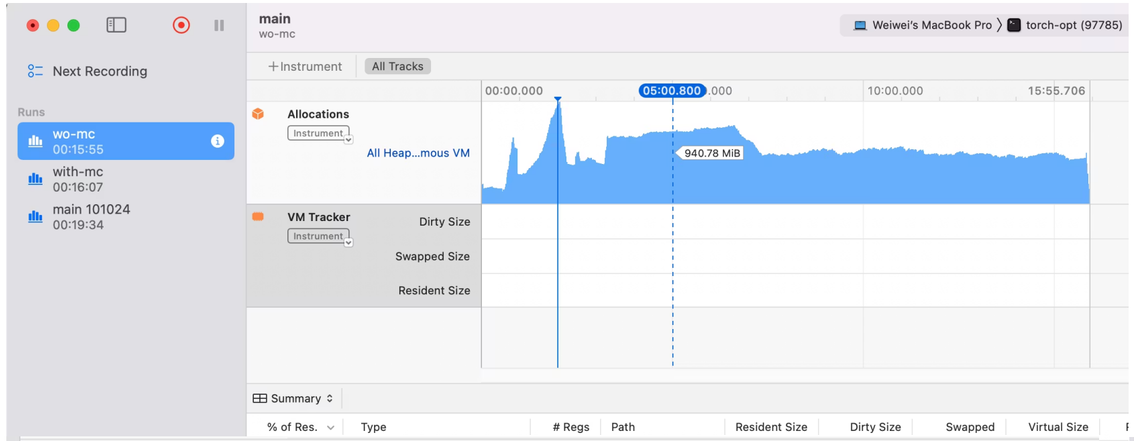
- Add *SetMachineFunctionBasePass* at the beginning of codegen pipeline to give MachineFunctions global numbering.
- Add *SyncX86SymbolTables* in AsmPrint pipeline to populate external MCSymbols to other llvm module split's MCContext so that they can be unique across all splits.
- Move MachineFunctions into AsmPrinter's MachineModuleInfo
  - Move private class member from off tree is hard.
- Workarounds for nameless variables (asan build)
  - *llvm::Linker* rename them but only local to the split.
  - Give these variable explicit name in mojo pipeline to avoid non-unique naming by *llvm::Linker.*



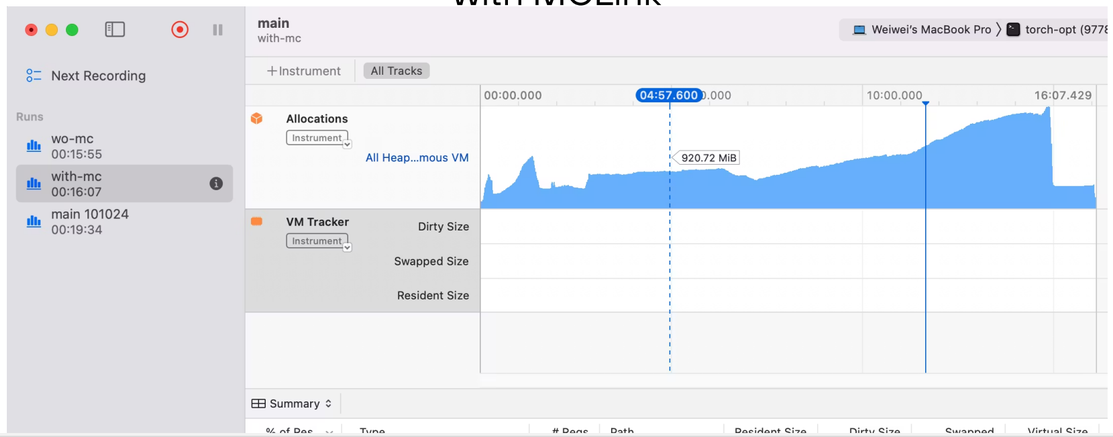
MCLink Implementation Details

Peak Memory Jump for running reset50-pytorch

without MCLink



with MCLink



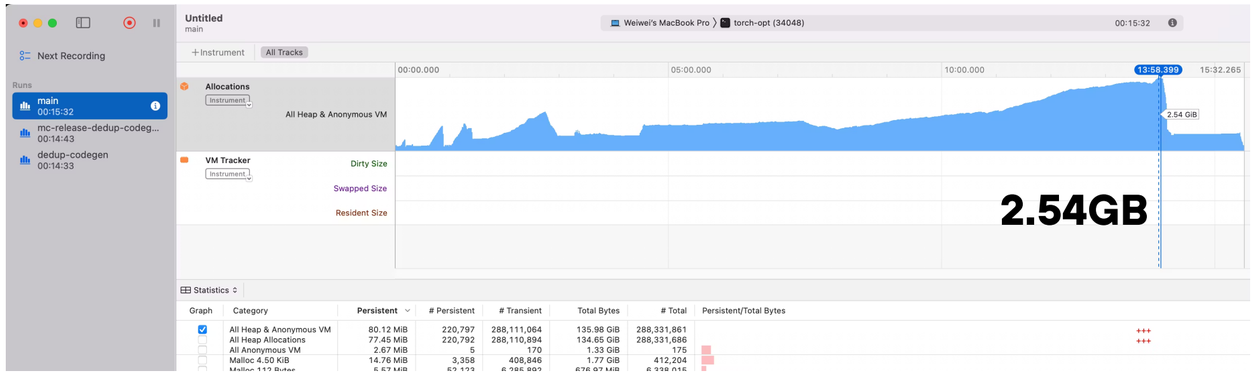
Statistics

Graph	Category	Persistent	# Persistent	# Transient	Total Bytes	# Total	Persistent/Total Bytes
☑	All Heap & Anonymous VM	114.19 MiB	207,242	124,362,379	69.61 GiB	124,569,621	
	All Heap Allocations	112.58 MiB	207,239	124,362,359	69.57 GiB	124,569,598	
	All Anonymous VM	1.61 MiB	3	20	40.94 MiB	23	
	Malloc 5.00 MiB	10.00 MiB	2	8	50.00 MiB	10	
	Malloc 6.50 MiB	6.50 MiB	1	5	39.00 MiB	6	
	Malloc 8.00 KiB	5.25 MiB	672	107,876	848.03 MiB	108,548	
	Malloc 16.00 KiB	5.25 MiB	336	34,807	549.11 MiB	35,143	
	Malloc 640.00 KiB	5.00 MiB	8	238	153.75 MiB	246	
	Malloc 1.25 MiB	5.00 MiB	4	130	167.50 MiB	134	
	Malloc 64.00 KiB	4.56 MiB	73	39,452	2.41 GiB	39,525	
	Malloc 4.00 KiB	4.55 MiB	1,165	544,316	2.08 GiB	545,481	
	Malloc 32.00 KiB	4.16 MiB	133	8,914	282.72 MiB	9,047	
	Malloc 96.00 KiB	3.47 MiB	37	10,931	1.00 GiB	10,968	
	llvm::MCSubtargetInfo	3.25 MiB	16	2,560	523.28 MiB	2,576	
	Malloc 3.25 MiB	3.25 MiB	1	44	146.25 MiB	45	
	Malloc 160.00 KiB	3.12 MiB	20	1,746	275.94 MiB	1,766	
	Malloc 1.00 KiB	2.98 MiB	3,047	23,505,236	22.42 GiB	23,508,283	
	Malloc 96 Bytes	2.63 MiB	28,703	5,144,573	473.63 MiB	5,173,276	
	Malloc 320.00 KiB	2.50 MiB	8	481	152.81 MiB	489	
	Malloc 48 Bytes	2.48 MiB	54,244	6,788,022	313.21 MiB	6,842,266	
	Malloc 832.00 KiB	2.44 MiB	3	140	116.19 MiB	143	
	Malloc 144 Bytes	2.11 MiB	15,400	1,942,176	268.83 MiB	1,957,576	
	Malloc 224 Bytes	2.06 MiB	9,649	286,968	63.36 MiB	296,617	
	Malloc 80 Bytes	2.00 MiB	26,176	30,512,438	2.28 GiB	30,538,614	
	Malloc 176 Bytes	1.67 MiB	9,966	478,562	82.00 MiB	488,528	
	Malloc 1.62 MiB	1.62 MiB	1	72	118.62 MiB	73	
	VM: Stack	1.61 MiB	3	4	3.80 MiB	7	
	Malloc 128.00 KiB	1.50 MiB	12	3,055	383.38 MiB	3,067	
	Malloc 160 Bytes	1.37 MiB	8,958	926,300	142.71 MiB	935,258	
	Malloc 23.00 KiB	1.28 MiB	57	46,083	1.01 GiB	46,140	
	Malloc 256.00 KiB	1.25 MiB	5	2,202	551.75 MiB	2,207	
	Malloc 1.50 KiB	1.13 MiB	772	11,735,104	16.79 GiB	11,735,876	
	Malloc 112 Bytes	1.01 MiB	9,446	1,529,130	164.34 MiB	1,538,576	
	llvm::LegalizerInfo	1.00 MiB	8	1,282	161.25 MiB	1,290	
	Malloc 11.50 KiB	1,000.50 KiB	87	56,722	637.99 MiB	56,809	
	llvm::ScalarEvolution::SC...	972.00 KiB	8	6,580	121.96 MiB	6,588	
	Malloc 3.00 KiB	885.00 KiB	295	256,985	753.75 MiB	257,280	
	Malloc 2.00 KiB	864.00 KiB	432	805,567	1.54 GiB	805,999	

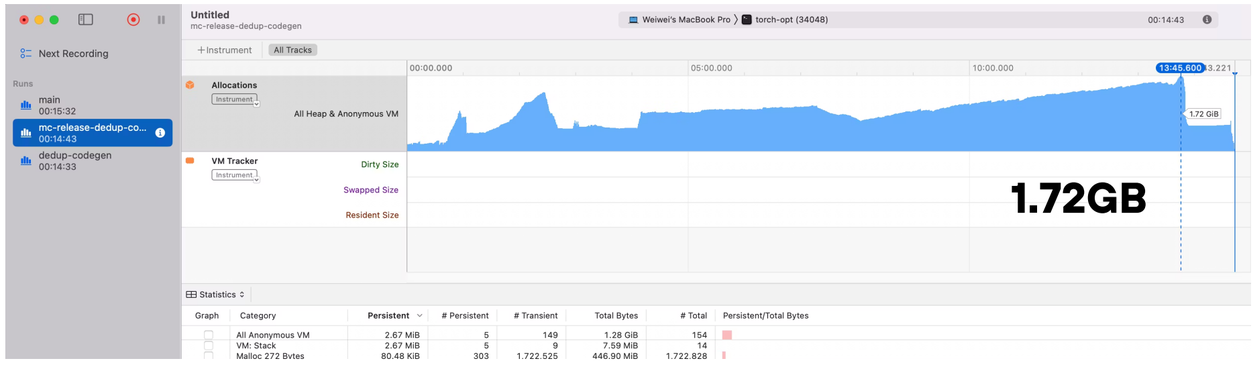
Graph	Category	Persistent	# Persistent	# Transient	Total Bytes	# Total	Persistent/Total Bytes
☑	All Heap & Anonymous VM	1.91 GiB	4,793,741	113,312,312	70.59 GiB	118,106,053	
	All Heap Allocations	1.91 GiB	4,793,738	113,312,292	70.55 GiB	118,106,030	
	All Anonymous VM	1.61 MiB	3	20	40.94 MiB	23	
	llvm::MCSubtargetInfo	485.76 MiB	2,392	0	485.76 MiB	2,392	
	Malloc 4.00 KiB	176.20 MiB	45,107	348,039	1.50 GiB	393,146	
	llvm::LegalizerInfo	149.38 MiB	1,195	0	149.38 MiB	1,195	
	Malloc 16.00 KiB	132.39 MiB	8,473	25,885	536.84 MiB	34,358	
	Malloc 144 Bytes	96.66 MiB	703,860	643,224	184.99 MiB	1,347,084	
	Malloc 8.00 KiB	78.34 MiB	10,027	107,825	920.72 MiB	117,852	
	Malloc 160 Bytes	52.26 MiB	342,489	301,461	98.26 MiB	643,950	
	Malloc 176 Bytes	47.77 MiB	284,623	73,623	60.13 MiB	358,246	
	Malloc 16.50 KiB	40.96 MiB	2,542	2,599	82.84 MiB	5,141	
	Malloc 224 Bytes	39.35 MiB	184,211	85,055	57.52 MiB	269,266	
	Malloc 32.00 KiB	38.41 MiB	1,229	7,694	278.84 MiB	8,923	
	Malloc 2.00 KiB	37.08 MiB	18,987	722,931	1.42 GiB	741,918	
	Malloc 80 Bytes	36.99 MiB	484,809	33,819,903	2.56 GiB	34,304,712	
	Malloc 112 Bytes	32.82 MiB	307,303	848,520	123.46 MiB	1,155,823	
	Malloc 96 Bytes	30.57 MiB	333,951	5,145,734	501.68 MiB	5,479,685	
	Malloc 128.00 KiB	27.00 MiB	216	2,089	288.12 MiB	2,305	
	Malloc 64.00 KiB	25.50 MiB	408	36,589	2.26 GiB	36,997	
	Malloc 512 Bytes	23.91 MiB	48,971	1,204,969	612.28 MiB	1,253,940	
	Malloc 48 Bytes	22.55 MiB	492,625	5,659,413	281.62 MiB	6,152,038	
	Malloc 800 Bytes	21.88 MiB	28,681	27,078	42.54 MiB	55,759	
	Malloc 1.00 MiB	21.00 MiB	21	20	41.00 MiB	41	
	Malloc 1.00 KiB	19.44 MiB	19,904	25,078,088	23.94 GiB	25,097,992	
	Malloc 32 Bytes	19.33 MiB	633,282	1,964,411	79.28 MiB	2,597,693	
	Malloc 256.00 KiB	17.75 MiB	71	1,919	497.50 MiB	1,990	
	Malloc 288.00 KiB	16.59 MiB	59	285	96.75 MiB	344	
	Malloc 16.00 MiB	16.00 MiB	1	0	16.00 MiB	1	
	Malloc 832.00 KiB	15.44 MiB	19	94	91.81 MiB	113	
	llvm::MCSectionMachO	14.02 MiB	3,588	0	14.02 MiB	3,588	
	Malloc 1.50 KiB	13.70 MiB	9,353	13,253,583	18.97 GiB	13,262,936	

MCLink Implementation Details

- Each split has its own llvm::TargetMachine due to mutable memory variables.
- SubtargetMap StringMap has many copies for each split, they are all identical and only one is needed for AsmPrint.
- Release memory early before AsmPrint to reduce peak memory footprint ASAP.
- Avoid codegen duplicated functions.



without early memory release



with early memory release

Mojo Compilation with Parallel LLVM

- Measured on c5n.metal
  - Turbo boost and hyper-threading disabled
  - CPU freq pinned at 2.9GHz.

* Changes symbol linkage type at subgaph level impacts optimization pipeline codegen quality + compilation time (LLVM can be unpredictable 😞).

** LLVM pipeline workload is smaller due to efforts on grinding down IR size.

test_matmul.mojo

par subgraph	par codegen	compilation time(s)	speedup	exe time (s)
no	no	36.35	1	26.78
yes*	no	34.37	1.06	38.95*
yes	yes	31.61	**1.15****	**26.72**

test_conv_epilogue.mojo

par subgraph	par codegen	compilation time(s)	speedup	exe time (s)
no	no	47.21	1	0.29
yes*	no	31.82	1.48	0.29*
yes	yes	37.61	**1.28****	**0.29**

Mojo Compilation with Parallel LLVM

- Measured on c5n.metal
  - Turbo boost and hyper-threading disabled
  - CPU freq pinned at 2.9GHz.

* Changes symbol linkage type at subgaph level impacts optimization pipeline codegen quality + compilation time (LLVM can be unpredictable 😞).

** LLVM pipeline workload is smaller due to efforts on grinding down IR size.

test_matmul.mojo

par subgraph	par codegen	compilation time(s)	speedup	exe time (s)
no	no	36.35	1	26.78
yes*	no	34.37	1.06	38.95*
yes	yes	31.61	**1.15****	**26.72**

test_conv_epilogue.mojo

par subgraph	par codegen	compilation time(s)	speedup	exe time (s)
no	no	47.21	1	0.29
yes*	no	31.82	1.48	0.29*
yes	yes	37.61	**1.28****	**0.29**

Mojo Compilation with Parallel LLVM

- Measured on c5n.metal
  - Turbo boost and hyper-threading disabled
  - CPU freq pinned at 2.9GHz.

* Changes symbol linkage type at subgaph level impacts optimization pipeline codegen quality + compilation time (LLVM can be unpredictable 😞).

** LLVM pipeline workload is smaller due to efforts on grinding down IR size.

test_matmul.mojo

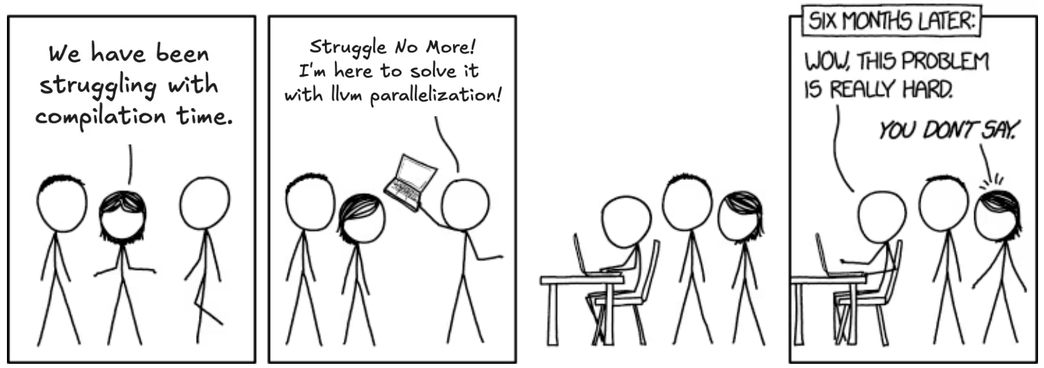
par subgraph	par codegen	compilation time(s)	speedup	exe time (s)
no	no	36.35	1	26.78
yes*	no	34.37	1.06	38.95*
yes	yes	31.61	**1.15****	**26.72**

test_conv_epilogue.mojo

par subgraph	par codegen	compilation time(s)	speedup	exe time (s)
no	no	47.21	1	0.29
yes*	no	31.82	1.48*	0.29*
yes	yes	37.61	**1.28****	**0.29**

Conclusions

- Parallelizing LLVM compilation helps improve overall mojo compilation time.
  - Mojo program is in one compilation unit.
  - MCLink helps to keep parallel llvm compilation with minimum side-effect.
  - Asynchronous Runtime for threading and dispatch parallel compilation tasks.
  - Build system (Bazel) be aware of mojo compilation threading, compiler server?
- Limitations
  - Only tested for X86 and AArch64.
  - Not applicable if codegen pipeline passes are inter-procedural, e.g. NVPTX backend.
  - Have to use some C++ tricks to work off-tree.
- We'd like to upstream this
  - llvm-module-splitter #121543
  - MCLinker #132989
- This works, but is this a good idea? 👀
  - Friend classes? Bytecode support for MIR? Make LLVM thread-safe? ThinLTO?



PC: xkcd - Here to Help

Questions

Acknowledgement:
The Mojo Language Team at Modular